Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 142 (2006) 129–141

www.elsevier.com/locate/entcs

Fast and Flexible Compression for Web Search Engines

Antonio Fariña[1], Nieves R. Brisaboa[2], Cristina París[3], and José R. Paramá[4]

Database Lab., Univ. da Coruña, Facultade de Informática
Campus de Elviña s/n, 15071 A Coruña, Spain.

Abstract

In this paper we present the adaptation of a compression technique, specially designed to compress large textual databases, to the peculiarities of web search engines.
The (s,c)-Dense Code belongs to a new category of compression techniques [17,9] that allows fast and flexible search directly on compressed files. However these methods are only suitable for large natural texts containing at least 1 megabyte, otherwise they would not achieve an attractive amount of compression.
In order to take advantage of the search capabilities of these techniques (they allow searches on compressed files up to eight times faster than searching on the plain versions [17]), we present a modification of the basic compression technique (s,c)-Dense Code to achieve reasonable compression ratios with small files, a requirement when we work with search engines.

Keywords: Compression, document repositories

1 Introduction

With the rapid growth of the World Wide Web in the last few decades, search engines have become essential tools for finding any kind of information.

⋆ Partially supported by Ministrio de Ciencia y Tecnología grants (#TIC2003-06593) and (#FIT-150500-2003-588); and by Xunta de Galicia grant(#PGIDIT02SIN10501PR).

[1] Email: fari@udc.es
[2] Email: brisaboa@udc.es
[3] Email: cparis@udc.es
[4] Email: parama@udc.es

doi:10.1016/j.entcs.2004.09.043

Many search engines have been developed, and some common examples include AltaVista [2], Alltheweb [1] and Google [4]. There also have been developed some search engines specialized in providing services to specific communities (countries for example). Some of these search engines are tumba! [19] (Portugal), todocl [6] (Chile), todobr [5] (Brazil), vieiros [7] (Galicia), buscopio [3] (Spain). These search engines are an alternative to global search engines for locating information within the pages of those communities.

Most of these search engines use their own technology, hardware architecture, indexing techniques, storage strategies, etc., developed by their own designers. However, some common characteristics can be found in many of these search engines.

Among those common characteristics, we can find that search engines store the original web pages in order to built the *snippet* (short sentence inside the page including the searched pattern) that will be shown in response to a query. In many cases web pages are stored compressed, generally with tools based on Ziv-Lempel techniques [20,21]. In addition, they usually employ an inverted index where each word has an associated pointer which points to the compressed file where the word appears.

In order to build the response to a user query, first the inverted index is used to retrieve the list of documents that match the query. Then, after using a ranking strategy to obtain those pages with the highest rank, it is necessary to uncompress the file and use a search algorithm to find the first occurrence of the searched words to build the snippet.

Obviously, the response to a user query should be provided as soon as possible, introducing serious time restrictions on the process described above. Therefore, although the compression of web pages is an attractive method to save space, if the compression scheme does not allow the search for words directly on the compressed text, the retrieval will be less efficient due to the necessity of decompression before the search.

Recently, several researches have developed compression techniques that allow the search for words in the compressed text without decompressing it, in such a way that the search can be up to eight times faster for certain queries [17,9]. However, these techniques are designed for large natural language texts containing at least 1 megabyte in order to achieve attractive compression ratios. This is unacceptable for many search engines, where each web page is compressed individually and therefore, the files that should be compressed contain only tens or hundreds of bytes.

In this paper we present an adaption of a compression technique called (s,c)-Dense Code [9,10] in order to overcome the problem of the file size. The (s,c)-Dense Code is a statistical semi-static method. This means that

the compression is achieved by replacing each original symbol with a unique codeword. However, in order to reconstruct the original information, the compressed file should include the equivalence between original symbols and codewords. This is the main drawback of this technique since for small files this information requires an extra space that ruins the compression ratio. The main contribution of this paper is a modification in the way this information is stored in order to improve the (s,c)-Dense Code compression ratio with small files.

On the other hand, by adapting (s,c)-Dense Code to small files we show that its benefits, specially its search speed, make this technique suitable for search engines, an important variable in these environments.

2 Compression techniques

Classic compression techniques, like the Huffman character oriented code [13], or the well-known algorithms of Ziv and Lempel [20,21], are not suitable for textual databases. One important disadvantage of these techniques is the inefficiency of direct search for words or other patterns on compressed texts. Besides, compression schemes based on Huffman codes are not often used with natural language texts because of the poor compression ratios achieved.

The idea of Huffman coding is to compress the text by assigning shorter codewords to more frequent symbols. Traditional implementations of Huffman code are character based, i.e., they use the characters as the symbols of the alphabet. In [15] they use the words in the text as the symbols to be compressed. From a compression point of view, character-based Huffman methods are able to reduce English text to approximately 60% of their original size, while word-based Huffman methods are able to reduce them to 25% of their original size, because the distribution of words is much more biased than the distribution of characters.

One of the most successful compression techniques that allows direct search on compressed texts is Tagged Huffman Code [17]. It uses sequences of bytes as codewords, where the first bit of each byte is reserved to flag whether the byte is the first byte of a codeword or not. Hence, only 7 bits of each byte are used for the Huffman code. Although this has a price in terms of compression performance, as the compressed file grows approximately by 11%, the addition of the tag bit permits direct search on the compressed text by simply compressing the pattern and then using any classical string matching algorithm.

The (s,c)-Dense Code [9,10] is a compression technique that has all the properties of the Tagged Huffman Code, that is: exact search for words

and phrases directly on the compressed text using any known sequential pattern matching algorithm, efficient word-based approximate and extended searches without any decompression, and efficient decompression of arbitrary portions of the text. However, it has some advantages over Tagged Huffman, among them, the most important for us are that the (s,c)-Dense Code obtains better compression ratios and that it is faster during the compression and decompression processes. It obtains compression ratios around 30% of the original text when it has at least 1 megabyte. This requirement of size is the only drawback of this technique, that in the case of its application to search engines becomes a serious problem since, as we have already noted, the compressed files are too small (only a few KBs). We will show later how we have tackled this issue.

This technique is a generalization of a previous compression technique called End-Tagged Dense Code [11,14] that obtains better compression ratios, as well as a simpler and faster encoding, than Tagged Huffman.

2.1 End-Tagged Dense Codes

The End-Tagged Dense Code [11] is based on Tagged Huffman, but instead of using the flag bit to signal the beginning of a codeword, the flag bit is used to signal the end of a codeword. That is, the flag bit is *0* for the first bit of any byte of a codeword except for the last one, which has a *1* in its flag bit.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code (no codeword is a prefix of another codeword), regardless of the contents of the other 7 bits. Therefore, there is no need at all to use Huffman coding over the remaining 7 bits, it is possible to use *all* the possible combinations, as long as the flag bit is used to signal the last byte of the codeword.

Consequently, the computation of codes is extremely simple: it is only necessary to sort the vocabulary words by decreasing frequency and then sequentially assign the codewords. That is, the first word is encoded as $\underline{1}0000000$, then second as $\underline{1}0000001$, until the 128^{th} as $\underline{1}1111111$. The 129^{th} word is coded as $\underline{0}0000000{:}\underline{1}0000000$, 130^{th} as $\underline{0}0000000{:}\underline{1}0000001$ and so on.

2.2 (s,c)-Dense Codes

The (s,c)-Dense Code is a statistical semi-static prefix code. It is statistical because the word frequency distribution in the original text is used to assign codewords to original words. It is semi-static because once a codeword is assigned to a word, this assignment do not change, that is, a word is encoded always with the same codeword.

The (s,c)-Dense Code codewords are formed by a sequence of symbols from an alphabet of 2^b values, to be precise, symbol values will be between 0 and 2^b-1. Due to implementation restrictions each symbol is a byte, that is, $b = 8$ and therefore a codeword is a sequence of bytes.

The idea is to compress the text by assigning a specific codeword to each original word in such a way that words with highest frequency have shorter codewords and vice versa. The process of codification is performed by assigning sequentially a codeword to each word in the original text with respect to its frequency order, that in the case of the most frequent words will consist of only 1 byte, and in descendent order of frequency, words will be coded with two, three, etc., bytes.

End-Tagged Dense Code (the precursor of (s,c)-Dense Code) uses 2^{b-1} values, from 0 to $2^{b-1}-1$, for all the bytes of a codeword except the last one (*continuers*), and uses the other 2^{b-1} values, from 2^{b-1} to 2^b-1, for the last byte of the codeword (*stoppers*).

However, for a given corpus with a specific word frequency distribution, it might be that a different number of *continuers* and *stoppers* could compress better than just using 2^{b-1} values for each group.

An (s, c)-Dense Code ($s + c = 2^b$) assigns codewords of one byte to the s most frequent words, two bytes to the next cs, three to the next c^2s, and so on. Note that sc^{k-1} codewords can be coded using k symbols. Digits between 0 and $c-1$ are called "continuers" and those between c and $c+s-1$ are called "stoppers".

It is clear that End-Tagged Dense Code is a (2^{b-1},2^{b-1})-Dense Code. A (s, c)-Dense Code can be made even more efficient by properly choosing the s and c values. Among all the possible sequences of continuers terminated by a stopper, for a given probability distribution the dense code is the one that minimizes the average codeword length.

In order to obtain the optimal s and c values for a given corpus, an algorithm was provided in [9,10]. It is outside the scope of this paper the description of this algorithm.

Example 2.1 *The codewords assigned to fifteen words by a (2,3)-Dense Code are as follows:* ⟨3⟩, ⟨4⟩, ⟨0,3⟩, ⟨0,4⟩, ⟨1,3⟩, ⟨1,4⟩, ⟨2,3⟩, ⟨2,4⟩, ⟨0,0,3⟩, ⟨0,0,4⟩, ⟨0,1,3⟩, ⟨0,1,4⟩, ⟨0,2,3⟩, ⟨0,2,4⟩, ⟨1,0,3⟩ *and* ⟨ 1,0,4 ⟩. *Obviously the codeword* ⟨3⟩ *will be assigned to the most frequent word,* ⟨4⟩ *to the second one and so on.*

Note that a codeword does not depend on the exact probability of the original word, but just on its frequency order. In fact, given a word rank i, we can obtain on the fly its codeword with a few operations, so we do not need

to store the code, just only the sorted vocabulary in the compressed file. The reverse operation can be performed as well, that is, given a codeword, one can obtain the rank position of the correspondent word in the sorted vocabulary.

Finally, a compressed file will be composed of two parts. The first one is the sorted vocabulary, that is to say, the list of words that appear in the original text ordered by frequency. The second part is the compressed text itself, where each word was replaced by its codeword.

3 (s,c)-Dense Code applied to search engines

3.1 Compressing

In this section, we will show how a search engine can take advantage of the (s,c)-Dense Code since, as we have already pointed out, by using (s,c)-Dense Code the search engine can search a pattern directly on the compressed files in order to build the snippet.

The compression process begins with the computation of the optimal s and c values using the appropriate algorithm [9,10]. Since around the optimal value of s for a given corpus, the compression is relatively insensitive to the exact value of s (and c) [9], it is possible to compute such values from a large collection of web pages harvested from the web and maintain this s value for any web page compressed in the future. That is, instead of computing s for each web page, the system will always use this value of s in order to save computation time in the compression process.

We performed some empirical studies using web pages extracted from the tumba! search engine. In those studies, it was found that $s = 166$ was the optimal value.

However if we computed the code for the complete corpus, such code would be too large (for example, in the case of the whole corpus of web pages of tumba!, we would have to code 10260417 codewords, one for each word found in the corpus). This would have two consequences: firstly, this vocabulary would not fit in memory, and therefore the compress/uncompres time would be slow, secondly, the compression achieved would be low. This situation is due to the size of the vocabulary, that forces words with low frequency to be coded with long codewords. This could be inappropriate for a specific web page where one word could have a high frequency for that page, but a low frequency with respect to the whole corpus, therefore it would be coded with a long codeword, losing compression ratio. This idea can be generalized to any word with low frequency, that is, due to the big amount of words found, on average they would be coded with long codewords.

In order to attenuate this problem, as the web is divided in many web

pages, each one with a different vocabulary and with a different frequency distribution and, since each one generates a compressed file, we decided to divide the words found in the whole collection in two different vocabularies: a ***common vocabulary*** shared by all files with those words with the highest frequency with respect to the whole corpus and, a ***specific vocabulary*** for each file with some of the words that do not appear in the *common vocabulary*, as can be seen in Figure 1.

One must keep in mind that the original (s,c)-Dense Code technique stores the list of words of each text within the compressed file to be able to reconstruct the initial information. With this small modification, we save space, as the *common vocabulary* is shared by all the files and, therefore in most cases we avoid the repetition of the words with highest frequency in the specific vocabulary.

It addition, not all the words that do not appear in the *common vocabulary* are placed in *specific vocabularies*. That is, given the list of words in a file, some of them belong to the *common vocabulary*, some of them are in the *specific vocabulary*, but some of them are not coded and therefore, these words are not present in the vocabularies. This is because it is not worth assigning a codeword to those words with frequency=1 and to those with frequency=2 and only one character, therefore these words remain uncompressed. This unusual situation should be signaled placing a special codeword at the beginning and at the end of the uncompressed words. Hence, if we observe Figure 1, the vocabulary needed to uncompress *ct3* can be obtained by linking together the common vocabulary (*cv*) and the specific vocabulary (*sv3*) and taking into account that some words are not coded.

After some experimental results to determine the best size of both vocabularies, it was found that the better results were obtained when the number of words in the common vocabulary plus the number of words in the specific vocabulary was up to the number of words that could be coded with two bytes (that is $s + sc$, approximately 15000 words).

3.2 Searching/Decompressing

Using the (s,c)-Dense Code, it is not necessary to uncompress the file to find a pattern. The only requirement is to find the codeword/s corresponding to the pattern to be searched (usually, several words in our case) and then use a conventional pattern matching algorithm, as it is explained below.

Let us suppose that we are searching for one word. In order to obtain the corresponding codeword, it is needed to find the pattern within the global vocabulary (the common and the specific vocabularies linked together). Supposing the common vocabulary is in main memory, the specific vocabulary

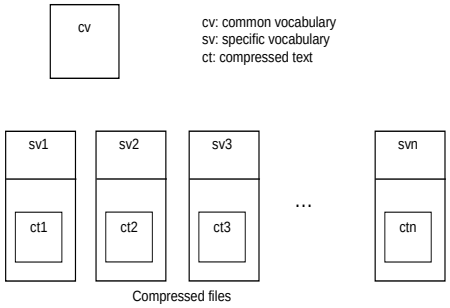

Fig. 1. Structure of compressed files.

can be located in main memory in $O(v)$, being v the size of the specific vocabulary. Thus, the codeword of the pattern can be found in $O(1)$ if the pattern is in the common vocabulary and in $O(v)$ if the pattern is in the specific vocabulary. Therefore, the search (of the codeword in the global vocabulary) can be performed in $O(v)$.

The common vocabulary is in main memory stored as a hash table, therefore the search of a codeword in the common vocabulary has a negligible cost. However, if the word is not in the common vocabulary, the specific one should be explored. This vocabulary is stored in the corresponding compressed file, hence it should be sought sequentially. In order to speed up future searches in the specific vocabulary, at the same time that is being sought, the specific vocabulary is stored in a hash table in main memory. In addition, as soon as the searched word is found in the vocabulary the seek ends, saving computation time. Future searches in the specific vocabulary can take advantage of the part of the specific vocabulary stored in the hash table, however if the new searched word is not present in the hash table, the sequential search in the compressed file should be continued (adding the new inspected codewords to the hash table).

Once we have the codeword/s corresponding to the searched pattern, we have to find the first occurrence of such codeword/s in the text in order to build the snippet. This search is performed by means of a classic pattern matching algorithm without any specific consideration, that is, considering the compressed text as plain text.

In general, classical pattern matching algorithms use a search window of the size of the pattern that is slid from left to right along the text, and the pattern is sought inside the window. The algorithms differ in the way in which the window is shifted and the pattern is searched.

We have chosen to use the Horspool algorithm [12], which is the one that

yields the best performance with a large alphabet [16], as that is the case since the 256 ASCII values are used as alphabet. With this approach the search is done backwards along the search window, reading the longest suffix of the window that is also a suffix of the pattern. This approach enables on average to avoid reading some characters of the text. The most famous algorithm using this technique is the Boyer-Moore [8] algorithm, which has been simplified by Horspool [12] and by Sunday [18].

The algorithm works as follows. For each position of the search window, its last character is compared with the last character of the pattern. If they match, the search window is verified backwards against the pattern until either the pattern is found or fail on a text character. Then, if the pattern does not match completely with the search window, whether there was a partial match or not, the window is shifted according to the next occurrence of its last character in the pattern.

Figure 2 shows an example using Horspool algorithm. In *step 1* the last character of the window (a) matches the last character of the pattern. The verification continues backwards, and it fails on the next character ($c \neq r$). Then, the window is shifted so that its last character (a) is aligned with the next occurrence of the same character in the pattern (shift = 3). In *step 2* there is a fail in the last character of the pattern ($b \neq a$) therefore the window is shifted according to its last character again (shift = 2). Finally, in *step 3* the last character of the window matches the last character of the pattern. The window is verified backwards and an occurrence is found.

4 Empirical results

We have chosen randomly some files from a collection harvested from the web by tumba! to perform the empirical studies. Tumba! current technology and (s,c)-Dense Code were compared in terms of compression ratio and the time to build the snippet.

We have used tumba! since we are collaborating with the development team of this search engine. This allow us to access to the components of their system and then we only have to construct the software that implements our algorithms in such a way that it replaces the compressing module in the tumba! system. After that, we can make the empirical studies with the real web pages harvested by tumba!.

In Table 1 we present the original file size in bytes, and the compression size with both methods, tumba! and (s,c)-Dense Code. The last column indicates the compression ratio obtained by both methods. One must take into account that the size of the specific vocabulary explained in Section 3 is added to the

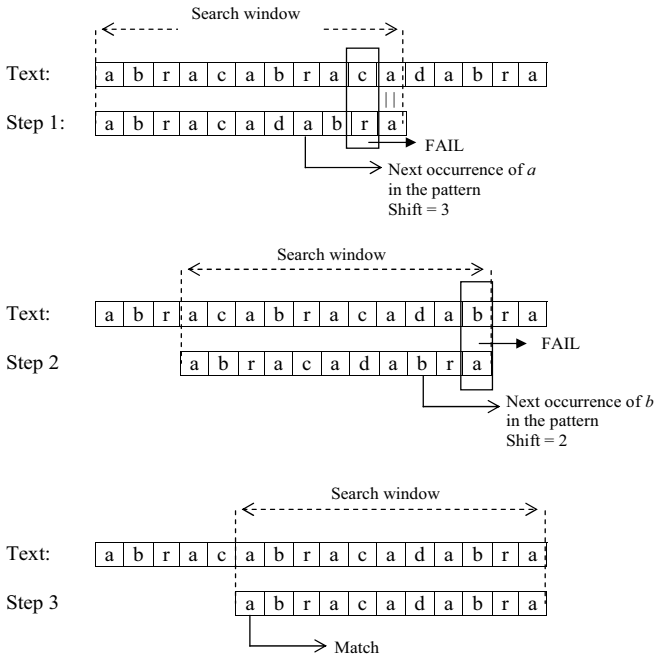

Fig. 2. Horspool example.

size of the compressed files using (s,c)-Dense Code.

Table 2 shows the time used to build the snippet by tumba! and by the new method using (s,c)-Dense Code as the compression technique and Horspool as the search algorithm. It can be seen that using (s,c)-Dense Code the time to build the snippet decreases significantly as there is no need to uncompress the file to search for the pattern within the file.

Observe that although (s,c)-Dense Code loses compression ratio when it is compared with tumba!, it reduces the time, which is an important variable in search engines in order to give a response to a user query as fast as possible. The modification of the (s,c)-Dense Code improves the results of the original version when it is applied to small files.

5 Conclusions and Future work

We have presented in this article an adaptation of a compression technique and a search algorithm as an alternative to those used by most search engines, that is, those based in Ziv-Lempel. Using (s,c)-Dense Code as the compression technique, direct search is possible on the compressed file saving the time needed to uncompress it.

However, despite the limitation of the (s,c)-Dense Code to compress small

File Name	Size (bytes)			compression ratio (%)	
	Original	Tumba!	(s,c)-DC	Tumba!	(s,c)-DC
1069.txt	581	340	291	58.52	50.09
1065.txt	1629	846	1229	51.93	75.45
1037.txt	2800	1369	1930	48.89	68.93
1291.txt	4939	2113	3284	42.78	66.49
1249.txt	5212	2324	3336	44.59	64.01
5573.txt	6644	3597	4552	54.14	68.51
3554.txt	7497	3900	4753	52.02	63.40
4408.txt	8226	3779	4570	45.94	55.56
2691.txt	9587	4517	5642	47.12	58.85
7101.txt	10305	4983	6576	48.36	63.81
135.txt	11253	4133	6013	36.73	53.43
4081.txt	11608	5133	6168	44.22	53.14
1358.txt	17668	6613	11461	37.43	64.87
2276.txt	165552	62602	85484	37.81	51.64
6428.txt	183179	72192	94205	39.41	51.43
6200.txt	188755	69242	84277	36.68	44.65
2161.txt	192884	71376	108493	37.00	56.25
7391.txt	269294	105524	134712	39.19	50.02
9916.txt	389547	148916	197238	38.23	50.63
2439.txt	551392	195271	226209	35.41	41.03

Table 1
Comparison of compression ratios.

files, a modification to tackle this problem was presented, adding a common vocabulary with those words with the highest frequency, that will be shared by all the files in the collection. This is the major contribution of this paper.

Some empirical results comparing the new method with tumba! were shown in terms of compression ratio and the time to build the snippet. The new technique does not improve tumba! when it comes to compression ratio, as the size of specific vocabulary (Section 3) is added to the size of the compressed file. On the other hand, the time needed to build the snippet decreases significantly when the new technique is used.

Despite the loss in compression ratio, the time reduction achieved makes these methods a suitable alternative to use with search engines, where response time is the most important variable. The compression ratio achieved by the modified (s,c)-Dense Code (around 50 %) is only 10 percent points worst than the tumba! ratios. However these results represent a good balance between search speed and compression ratio when we deal with search engines.

File Name	Time (ms)		Time decrease (%)
	Tumba	(s,c)-DC	
1069.txt	0.622	0.120	80.71
1065.txt	0.602	0.100	83.39
1037.txt	1.302	0.260	80.03
1291.txt	1.802	0.260	85.57
1249.txt	2.182	0.400	81.67
5573.txt	3.424	0.600	82.48
3554.txt	2.702	0.582	78.46
4408.txt	4.006	0.662	83.47
2691.txt	4.228	0.680	83.92
7101.txt	5.588	1.040	81.39
135.txt	2.424	0.240	90.10
4081.txt	5.448	0.840	84.58
1358.txt	3.164	0.500	84.20
2276.txt	4.046	0.540	86.65
6428.txt	1.302	0.182	86.02
6200.txt	1.862	0.240	87.11
2161.txt	2.844	0.442	84.46
7391.txt	1.182	0.720	39.09
9916.txt	1.200	0.220	81.67
2439.txt	2.022	1.642	18.79

Table 2
Time comparison.

Until now, the exact pattern is searched within the compressed text. In the future, these algorithms should be improved to allow the search for different versions of the pattern, for example, there should be a match whether the pattern is written in capital letters or in lower-case letters, with accents or not, etc.

References

[1] *Alltheweb.com*, http://www.alltheweb.com.

[2] *Altavista*, http://www.altavista.com.

[3] *Buscopio: Directorio y buscador de buscadores*, http://www.buscopio.net.

[4] *Google*, http://www.google.com.

[5] *Todobr: Todo brasil na internet*, http://www.todobr.com.br.

[6] *Todocl: El buscador de todo chile*, http://www.todocl.cl.

[7] *Vieiros: Buscador*, http://buscador.vieiros.com/ligazons/buscador.

[8] Boyer, R. S. and J. S. Moore, *A fast string searching algorithm*, Communications of the ACM **20** (1977), pp. 762–772, see also [18].

[9] Brisaboa, N., A. Fariña, G. Navarro and M. Esteller, *(s,c)-dense coding: An optimized compression code for natural language text databases*, in: *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, 2003, pp. 122–136.

[10] Brisaboa, N., A. Fariña, G. Navarro, E. L. Iglesias, J. R. Paramá and M. Esteller, *Codificación (s,c)-densa: optimizando la compresión de texto en lenguaje natural*, in: *VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2003)*, 2003, pp. –.

[11] Brisaboa, N., E. Iglesias, G. Navarro and J. Paramá, *An efficient compression code for text databases*, in: *25th European Conference on IR Research, ECIR 2003*, LNCS 2633, 2003, pp. 468–481.

[12] Horspool, R. N., *Practical fast searching in strings*, Software Practice and Experience **10** (1980), pp. 501–506.

[13] Huffman, D. A., *A method for the construction of minimum-redundancy codes.*, Proc. Inst. Radio Eng. **40** (1952), pp. 1098–1101.

[14] Iglesisas, E., N. Brisaboa, J. Paramá, A. Fariña, G. Navarro and M. Esteller, *Usando técnicas de compresión de textos en bibliotecas digitales*, in: *IV Jornadas de Bibliotecas Digitales (JBIDI 2003)*, 2003, pp. 39–48.

[15] Moffat, A., *Word-based text compression*, Software - Practice and Experience **19** (1989), pp. 185–198.

[16] Navarro, G. and M. Raffinot, "Flexible pattern matching in Strings," Cambridge University Press, 2002.

[17] Silva de Moura, E., G. Navarro, N. Ziviani and R. Baeza-Yates, *Fast and flexible word searching on compressed text*, ACM Transactions on Information Systems **18** (2000), pp. 113–139.

[18] Sunday, D. M., *A very fast substring search algorithm*, Communications of the ACM **33** (1990), pp. 132–142, see also [8].

[19] XLDB, *Tumba! - temos um motor de busca alternativo*, http://www.tumba.pt.

[20] Ziv, J. and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory **23** (1977), pp. 337–343.

[21] Ziv, J. and A. Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Transactions on Information Theory **24** (1978), pp. 530–536.